The Story So Far...

- Looking at simple **client/server** interaction, and use of **remote procedure call** (RPC)
  - invoking methods on server over the network
  - middleware generates stub code which can **marshal / unmarshal** arguments and replies
  - saw case study of NFS (RPC-based file system)
- In the 1990s started to see OOM
  - Object-oriented middleware (CORBA, DCOM, ...)
  - Extends RPC model to remote objects
Java RMI

• 1995: Sun extended Java to allow RMI
  – RMI = Remote Method Invocation
• Essentially an OOM scheme for Java with clients, servers and an object registry
  – object registry maps from names to objects
  – supports bind()/rebind(), lookup(), unbind(), list()
• RMI was designed for Java only
  – no goal of OS or language interoperability
  – hence cleaner design, tighter language integration
  – E.g., distributed garbage collection

RMI: New Classes

• remote class:
  – one whose instances can be used remotely
  – within home address space, a regular object
  – within foreign address spaces, referenced indirectly via an object handle
• serializable class: [nothing to do with transactions!]
  – object that can be marshalled/unmarshalled
  – if a serializable object is passed as a parameter or return value of a remote method invocation, the value will be copied from one address space to another
  – (for remote objects, only the object handle is copied)
RMI: New Classes

• **remote class:**
  - one whose instances can be used remotely
  - within home address space, a regular object
  - within foreign address spaces, referenced indirectly via an object handle

• **serializable class:**
  - object that can be marshalled/unmarshalled
  - if a serializable object is passed as a parameter or return value, when deserialization, the value will be copied from one address space to another
  - (for remote objects, only the object handle is copied)

RMI: The Big Picture

- Registry can be on server... or one per distributed system
  - client and server can find it via the `LocateRegistry` class
- Objects being serialized are annotated with a URL for the class
  - unless they implement `Remote` => replaced with a remote reference
Distributed Garbage Collection

- With RMI, can have local & remote object references scattered around a set of machines
- Build distributed GC by leveraging local GC:
  - When a server exports object O, it creates a skeleton S[O]
  - When a client obtains a remote reference to O, it creates a proxy object P[O], and remotely invokes **dirty(O)**
  - Local GC will track the liveness of P[O]; when it is locally unreachable, client remotely invokes **clean(O)**
  - If server notices no remote references, can free S[O]
  - If S[O] was last reference to O, then it too can be freed
- Like DCOM, server removes a reference if it doesn't hear from that client for a while (default 10 mins)

---

OOM: Summary

- OOM enhances RPC with objects
  - types, interfaces, exceptions, ...
- Seen CORBA, DCOM and Java RMI
  - All plausible, and all still used today
  - CORBA most general (language and OS agnostic), but also the most complex: design by committee
  - DCOM is MS-only; being phased out for .NET
  - Java RMI decent starting point for simple distributed systems... but lacks many features
  - (EJB is a modern CORBA/RMI/<stuff> megalith)
XML-RPC

• Systems seen so far all developed by large industry, and work fine in the local area...
  – But don’t (or didn’t) do well through firewalls ;-)
• In 1998, Dave Winer developed XML-RPC
  – Use XML to encode method invocations (method names, parameters, etc)
  – Use HTTP POST to invoke; response contains the result, also encoded in XML
  – Looks like a regular web session, and so works fine with firewalls, NAT boxes, transparent proxies, ...

XML-RPC Example

<table>
<thead>
<tr>
<th>XML-RPC Request</th>
<th>XML-RPC Response</th>
</tr>
</thead>
</table>
| <?xml version="1.0"?>
<methodCall>
<methodName>util.IntToString</methodName>
<params>
<param>
<value><i4>55</i4></value>
</param>
</params>
</methodCall> | <?xml version="1.0"?>
<methodResponse>
<params>
<param>
<value><string>Fifty Five</string></value>
</param>
</params>
</methodResponse> |

• Client side names method (as a string), and lists parameters, tagged with simple types
• Server receives message (via HTTP), decodes, performs operation, and replies with similar XML
• Inefficient & weakly typed... but simple, language agnostic, extensible, and eminently practical!
SOAP & Web Services

• XML-RPC was a victim of its own success
• WWW consortium decided to embrace it, extend it, and generally complify it up
  – SOAP (Simple Object Access Protocol) is basically XML-RPC, but with more XML bits
  – Support for namespaces, user-defined types, multi-hop messaging, recipient specification, ...
  – Also allows transport over SMTP (!), TCP & UDP
• SOAP is part of the Web Services world
  – As complex as CORBA, but with more XML ;-)  

Moving away from RPC

• SOAP 1.2 defined in 2003
  – Less focus on RPC, and more on moving XML messages from A to B (perhaps via C & D)
• One major problem with all RPC schemes is that they were synchronous:
  – Client is blocked until server replies
  – Poor responsiveness, particularly in wide area
• 2006 saw introduction of AJAX
  – Asynchronous Javascript with XML
  –Chief benefit: can update web page without reloading
• Examples: Google Maps, Gmail, Google Docs, ...
REST

• AJAX still does RPC (just asynchronously)
• Is a procedure call / method invocation really the best way to build distributed systems?
• **Representational State Transfer** (REST) is an alternative ‘paradigm’ (or a throwback?)
  – Resources have a name: URL or URI
  – Manipulate them via PUT (insert), GET (select), POST (updated) and DELETE (delete)
  – Send state along with operations
• Very widely used today (Amazon, Flickr, Twitter)

Client-Server Interaction: Summary

• Server handles requests from client
  – Simple request/response protocols (like HTTP) useful, but lack language integration
  – RPC schemes (SunRPC, DCE RPC) address this
  – OOM schemes (CORBA, DCOM, RMI) extend RPC to understand objects, types, interfaces, exns, ...
• Recent WWW developments move away from traditional RPC/RMI:
  – Avoid explicit IDLs since can slow evolution
  – Enable asynchrony, or return to request/response
Clocks and distributed time

• Distributed systems need to be able to:
  – order events produced by concurrent processes;
  – synchronize senders and receivers of messages;
  – serialize concurrent accesses to shared objects; and
  – generally coordinate joint activity
• This can be provided by some sort of “clock”:
  – physical clocks keep time of day
    • (must be kept consistent across multiple nodes – why?)
  – logical clocks keep track of event ordering
• Relativity can’t be ignored: think satellites

Physical Clock Technology

• Quartz Crystal Clocks (1929)
  – resonator shaped like a tuning fork
  – laser-trimmed to vibrate at 32,768 Hz
  – standard resonators accurate to 6ppm at 31°C... so will gain/lose around 0.5 seconds per day
  – stability better than accuracy (about 2s/month)
  – best resonators get accuracy of ~1s in 10 years
• Atomic clocks (1948)
  – count transitions of the cesium 133 atom
  – 9,192,631,770 periods defined to be 1 second
  – accuracy is better than 1 second in 6 million years...
Coordinated Universal Time (UTC)

- Physical clocks provide ‘ticks’ but we want to know the actual time of day
  - determined by astronomical phenomena
- Several variants of universal time
  - **UT0**: mean solar time on Greenwich meridian
  - **UT1**: UT0 corrected for polar motion; measured via observations of quasars, laser ranging, & satellites
  - **UT2**: UT1 corrected for seasonal variations
  - **UTC**: civil time, tracked using atomic clocks, but kept within 0.9s of UT1 by occasional leap seconds

Computer Clocks

- Typically have a real-time clock
  - CMOS clock driven by a quartz oscillator
  - battery-backed so continues when power is off
- Also have range of other clocks (PIT, ACPI, HPET, TSC, ...), mostly **higher frequency**
  - free running clocks driven by quartz oscillator
  - mapped to real time by OS at boot time
  - programmable to generate interrupts after some number of ticks (≈ some amount of real time)
Operating system use of clocks

- OSes use time for many things
  - Periodic events – e.g., time sharing, statistics, at, cron
  - Local I/O functions – e.g., peripheral timeouts; entropy
  - Network protocols – e.g., TCP DELACK, retries, keep-alive
  - Cryptographic certificate/ticket generation, expiration
  - Performance profiling and sampling features
- “Ticks” trigger interrupts
  - Historically, timers at fixed intervals (e.g., 100Hz)
  - Now, “tickless”: timer reprogrammed for next event
  - Saves energy, CPU resources – especially as cores scale up

The Clock Synchronization Problem

- In distributed systems, we’d like all the different nodes to have the same notion of time, but
  - quartz oscillators oscillate at slightly different frequencies (time, temperature, manufacture)
- Hence clocks tick at different rates:
  - create ever-widening gap in perceived time
  - this is called clock drift
- The difference between two clocks at a given point in time is called clock skew
- Clock synchronization aims to minimize clock skew between two (or a set of) different clocks
Clock Skew and Clock Drift

08:00:00  
February 18, 2012  
08:00:00

NB: Steve Hand’s watches, not mine.

08:01:24  
March 23, 2012  
08:00:00

Skew = 84 seconds  
Drift = 84s / 34 days  
= +2.47s per day

08:01:48

Skew = 108 seconds  
Drift = 108s / 34 days  
= +3.18s per day
Next time

• More on physical time
  – Sources of global time information
  – Various algorithms for time synchronisation
  – The Network Time Protocol (NTP)

• Ordering
  – The “happens-before” relation
  – Lamport clocks
  – Vector clocks